

Lecture 8: Arrays in JavaScript

Written by Ben Yan, from materials by Jerry Cain

Creating Arrays

An array is an ordered collection of individual data values. The values inside are called **elements**, the number of elements is called its **length**, and like strings, arrays have a **length** property you can access.

Each element is identified by its position within the array, which is referred to as its **index**. In JavaScript, like all sensical programming languages ☺, the indices begin with **0** and extend up to **length - 1**.

The simplest way to manufacture an array in JavaScript is to patiently and explicitly **list its elements separated by commas, and surrounded by square brackets** on both ends like bunny ears. For instance,

```
let coinValues = [1, 5, 10, 25, 50, 100];
```

creates an array with the standard coins in U.S. currency. I like to draw arrays as numbered boxes, e.g.,

element	1	5	10	25	50	100
index	0	1	2	3	4	5

Unlike strings, arrays can be modified in memory! Suppose that, in 2011, amid the debt-ceiling crisis, President Obama had gone through with a (real) plan to mint & issue 1-trillion-dollar coins. In this case, `coinValues.push(1e12)` would result in an updated array `[1, 5, 10, 25, 50, 100, 1e12]`.

Sum Array

This function accepts an **array** as input, adds up the elements, and returns the sum. Take note of the loop pattern expressed here—cycling through each of the array's elements in turn. It'll appear a lot ☺.

```
function sumArray(array) {
  let sum = 0;
  for (let i = 0; i < array.length; i++) {
    sum += array[i]; // accesses the element at index i
  }
  return sum;
}
```

Index Array

This function creates an array by accretion, one element a time. The returned value is an array of **n** elements equal to each of their indices, e.g., `createIndexArray(5)` returns `[0, 1, 2, 3, 4]`.

```
function createIndexArray(n) {
  let array = [];
  for (let i = 0; i < n; i++) {
    array.push(i);
  }
  return array;
}
```

Make Change

We'll write our first marquee function for arrays, **makeChange**, which given a **change** (debt) in cents, displays the number of coins of each type necessary to produce that change—using the coin values in constant arrays **COIN_VALUES** and **COIN_NAMES**. For the denominations available in U.S. currency,

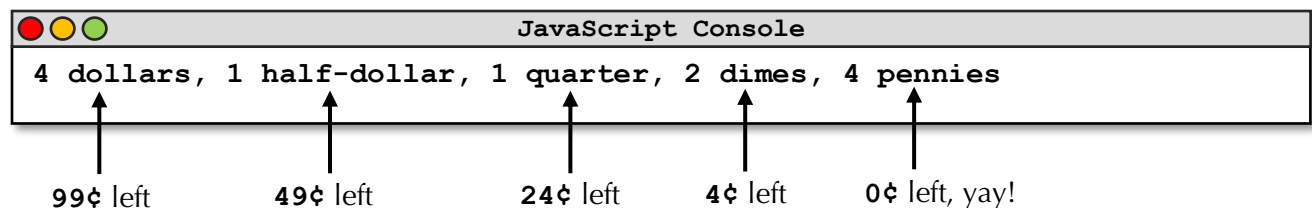
```
const COIN_VALUES = [1, 5, 10, 25, 50, 100];
const COIN_NAMES = ["penny", "nickle", "dime", "quarter",
                    "half-dollar", "dollar"];
```



We assume the currency is designed so that this (greedy) strategy always produces the correct result.

- Start with the last, most expensive element in the array (e.g., "**dollar**"), and use as many as those as possible. For instance, if **change** = **499¢**, we'd use **4** dollars, leaving **99¢** in change.
- Move on to the previous element, and give as many of those as possible (e.g., for **99¢**, use **1** half-dollar), continuing this banking process until you hit rock bottom at the number of pennies.

The following facsimile of a **JavaScript Console** displays the desired output for **change** = **499**.



You may assume a fully functional **createRegularPlural**, which accepts a coin name (or any **word** really), and forms its grammatically correct plural form, e.g., for "**penny**", we'd return "**pennies**".

First, for some remaining **change**, and a coin index **i** in the arrays above, the maximum number of those coins that can be used is given by this formula. Note with **floor**, the fractional part is discarded.

```
let n = Math.floor(change / COIN_VALUES[i]);
```

For instance, **Math.floor(499/100) => Math.floor(4.99) => 4**, if we're using dollar coins. Next, we use a non-zero number of these coins, we report it in string form, e.g., "**2 nickels**" (like the Dr. Doofenschmirtz meme). To do this, we need the coin's name, pluralizing it if more than one is used.

```
let noun = COIN_NAMES[i];
if (n > 1) noun = createRegularPlural(noun);
```

We then stringify the coin name and its quantity, separating it from other coins with a comma.

```
if (result.length > 0) result += ", "; // does not apply for 1st coin used
result += n + " " + noun;
```

Our implementation of **makeChange** is written below. We start from the highest coin, and repeat the steps above until all coins are accounted for, and so their accumulation fully compensates for **change**.

```
function makeChange(change) {
  let result = "";
  for (let i = COIN_VALUES.length - 1; i >= 0; i--) {
    let n = Math.floor(change / COIN_VALUES[i]);
    if (n > 0) {
      // put in string form how many of this coin were used
      let noun = COIN_NAMES[i];
      if (n > 1) noun = createRegularPlural(noun);

      if (result.length > 0) result += ", ";
      result += n + " " + noun; // e.g. "5 nickels"

      change -= n * COIN_VALUES[i]; // reduce remaining change
    }
  }
  return result;
}
```

Bonus: Pluralize Word

This is far from mandatory reading, as it pertains more to grammar than to arrays. But if you're curious, to implement the `createRegularPlural` function used above, we can generally follow these three rules for noun pluralization. There are lingering exceptions, of course, as English is 🙌 complicated 🙌, like *tooth* and *teeth*, *child* and *children*, *datum* and *data*. But you don't have to worry about them 😊.

- If the word ends in "s", "x", "z", "ch", or "sh", add "es" to the word. For instance, "starfish" -> "starfishes", and "matrix" -> "matrices".
- If the word ends in "y" preceded by a consonant, change the "y" to "ies". For instance, "berry" -> "berries", "sky" -> "skies", and "ivy" -> "ivies".
- In all other cases, just add an "s" 😊. For instance, "computer" -> "computers".

Here's our implementation of `createRegularPlural`, which uses a helper function defined below.

```
function createRegularPlural(word) {
  if (word.endsWith("s") || word.endsWith("x") || word.endsWith("z") ||
      word.endsWith("ch") || word.endsWith("sh")) {
    return word + "es";
  } else if (word.endsWith("y") && word.length > 1 &&
             isEnglishConsonant(word.charAt(word.length - 2))) {
    return word.substring(0, word.length - 1) + "ies";
  } else {
    return word + "s";
  }
}

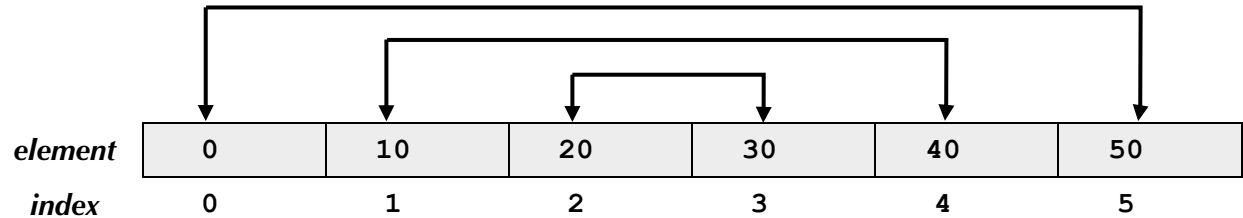
function isEnglishConsonant(ch) { // uses helper functions from Lecture 7
  return isLetter(ch) && !isEnglishVowel(ch);
}
```

Note we use the method `endsWith`, which returns `true` if the string ends with the supplied suffix.

Reverse Cursed Technique

Our last program for today **reverses the elements of an array in-place**, without creating a new one. This relies on the JavaScript principle that, when you pass an array into a function, e.g., `reverseArray`, only a **reference** to the array is actually shared, without birthing a separate copy. The effect is that **when a function changes an element of an array passed-in, that change will persist after the function returns**.

Suppose `arr = [0, 10, 20, 30, 40, 50]`, i.e. each element equals its index multiplied by 10.



Calling `reverseArray(arr)` should result in the array `arr` being updated to, no surprise, its reverse.

<i>element</i>	50	40	30	20	10	0
<i>index</i>	0	1	2	3	4	5

Note this “**reverse cursed technique**” is achieved by **swapping pairs of elements**, starting from the first and last, then the second and second last, and pinching inward, as illustrated with the arrows above.

Keeping this in mind, we’ll write a helper function that accepts an **array**, and two of its indices **lh** and **rh**, as input, and **swaps just the pair of elements at those indices**. That is, `array[lh]` becomes `array[rh]`, and `array[rh]` becomes the **original value** of `array[lh]`. To memorize this original value, before it gets overridden, we’ll need a temporary variable `tmp`.

```
function swap(array, lh, rh) {
  let tmp = array[lh];
  array[lh] = array[rh];
  array[rh] = tmp;
}
```



As a visual metaphor, try swapping the contents in two glasses of juice. You can’t do it without a third!

To implement `reverseArray`, we iterate through each pair of elements opposite across the array, and swap them, e.g., for the `arr` above, we loop over `lh = 0` and `rh = 5`, `lh = 1` and `rh = 4`, so on.

```
function reverseArray(array) {
  for (let lh = 0; lh < array.length / 2; lh++) {
    let rh = array.length - lh - 1;
    swap(array, lh, rh);
  }
}
```

Note if the length is odd, the loop ends just before we reach the element at index `array.length / 2`. But this is totally ok! The center element is unaffected by the array’s reversal, so no swaps are needed 😊.

Methods

Each of these methods has their uses cases and starring moments to shine, but the three most common operations—and the ones you should prioritize and commit to memory—are **push**, **pop**, and **shift**.

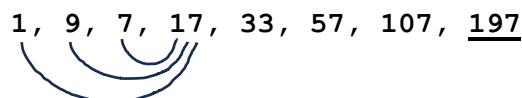
JavaScript Array Methods					
push(element, ...);					
Adds one or more elements to the end of the array, with a variable number of input parameters, e.g.,					
let arr = [];	element	1	2	4	8
arr.push(1);					
arr.push(2, 4, 8, 16);	index	0	1	2	3
pop()					
Removes and returns the last element of the array.					
shift()					
Removes and returns the first element of the array.					
unshift(element, ...);					
Adds one or more elements to the front of the array, with a variable number of input parameters.					
slice(start, finish)					
Returns a subarray beginning at the start index, and ending just before the finish index.					
splice(index, count, ...);					
Removes count elements starting at index , and optionally adds new elements. If you'd prefer to add new elements without removing any old ones, you can specify count as 0 , e.g., the method call arr.splice(i, 0, newElem) ; inserts newElem so that it now occupies index i of the array.					
concat(array, ...)					
Returns a new array that consists of all the input arrays concatenated together end-to-end, e.g.,					
let arrOne = [10, 30], arrTwo = [20, 40];					
let merged = concat(arrOne, arrTwo); // merged is array [10, 30, 20, 40]					
indexOf(element)					
Returns the first index at which element appears in the array, or -1 if not found.					
lastIndexOf(element)					
Returns the last index at which element appears in the array, or -1 if not found.					
reverse()					
Reverses the elements of the array in place. It does not return a new array but modifies the original.					
let arr = [1, 5, 10, 25, 50, 100];					
arr.reverse(); // arr is now [100, 50, 25, 10, 5, 1]					
sort()					
Sorts the elements of the array in ascending order, and modifies the original array directly.					

Keith Numbers

The last example covered today was a treatise on ✨ **Keith numbers** ✨. Such a number **num** happens to appear in a Fibonacci-like sequence with the initial terms based on **num**'s own digits, listed left to right, with each new term generating by summing the latest **n** terms, where **n** is the number of digits in **num**.

For example, **197** is a Keith number, and here's proof in the generated sequence from **[1, 9, 7]**.

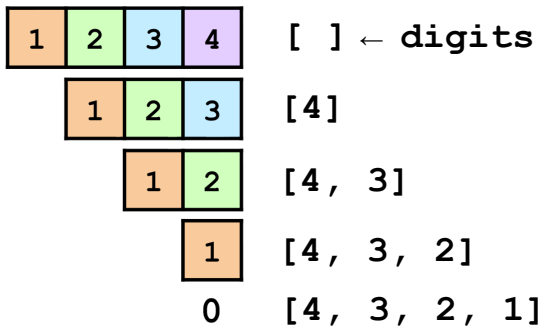
1, 9, 7, 17, 33, 57, 107, 197



Note that 197 appears in its own sequence ☺, where the first 3 digits $1 + 9 + 7$ generates 17, and $9 + 7 + 17$ generates 33, and so on. We'll write a function **isKeithNumber(num)**, which returns **true** if and only if **num** satisfies the Fibonacci-esque property above, and otherwise returns **false**.

First, we write a helper function that, given an integer **num**, returns an array of its digits left-to-right.

```
function createDigitsArray(num) {
  let digits = [];
  while (num > 0) {
    let digit = num % 10;
    digits.push(digit);
    num = Math.floor(n/10);
  }
  digits.reverse();
  return digits;
}
```



For instance, **createDigitsArray(1234)** returns the array **[1, 2, 3, 4]**. Note the digit-sum loop iterates over the digits from right-to-left, but **reverses at the very end to attain the correct ordering**.

Then, we write the actual **isKeithNumber** function, which can be implemented like this ☺.

```
function isKeithNumber(num) {
  if (n <= 0) return false;

  let partials = createDigitsArray(num); // stores the latest n terms
  while (partials[partials.length - 1] < num) {
    let sum = sumArray(partials);
    partials.push(sum); // adds the newest term
    partials.shift(); // removes the first / oldest term
  }

  return partials[partials.length - 1] === num;
}
```

From the **partials** array, initially just the digits, we generate the next number using the **sumArray** function from earlier, and add it to the array. However, since we only want to sum the latest **n** terms, we chomp off the first element in **partials**, so that it acts as a “sliding window” that contains only the most recent **n** elements, e.g., if **num = 1234**, so **n = 4**, the array starts as **[1, 2, 3, 4]**, becomes **[2, 3, 4, 10]**, then **[3, 4, 10, 19]**, and so on. Lastly, when we encounter a value at or above **num**, we exit the loop, and return **true** if we've landed exactly on **num** ✓, and **false** if we went beyond it ✗.